AD-A134 698   HIERARCHICAL CACHE ACCUMULATORS FOR SEQUENTIAL MODE   1/1
               ESTIMATION(U) ROCHESTER UNIV NY DEPT OF COMPUTER
               SCIENCE C M BROWN JUL 83 TR-125 N00014-78-C-0164
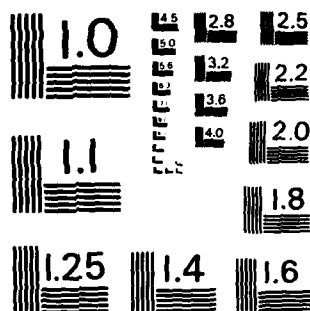UNCLASSIFIED   F/G 9/2   NL

END
DATE
FILMED
12
DTIC

MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS-1963-A

HIERARCHICAL CACHE ACCUMULATORS
FOR SEQUENTIAL MODE ESTIMATION

Christopher M. Brown
Computer Science Department
University of Rochester
Rochester, NY 14627 U.S.A.

TR 125

July 1983

HIERARCHICAL CACHE ACCUMULATORS
FOR SEQUENTIAL MODE ESTIMATION

Christopher M. Brown
Computer Science Department
University of Rochester
Rochester, NY 14627 U.S.A.

TR 125

July 1983

Abstract

A quad-tree-like data structure can be implemented with a content-addressable cache. The resulting structure captures spatial relations at several resolutions. Spatial relations such as containment and contiguity may be useful in flushing Cache-Hough Transform accumulators. Algorithms for accumulator cache management may be written that are related to some proposed in the literature for statistical mode estimation.

Key Words: Hough transform, histogram peak-finding, mode estimation, cache, resolution pyramids.

DTIC
ELECTE
NOV 1 4 1983
S D

B

1. Background and Overview

The Hough Transform (HT) is a parameter estimation strategy based on the statistical mode [Duda and Hart 1972; Ballard 1981]. More common strategies such as least squared error fitting (e.g. linear regression) are based on the statistical mean. The HT has achieved engineering importance in several areas of image understanding. It is an efficient implementation of generalized matched filter (template matching) detection, and its mode-based nature makes it highly noise-resistant. Outliers do not affect it, whereas they always have some effect on simple regression (of course, much research in statistics is concerned with rejecting outliers). HT is an important technique in some massively parallel computing architectures [Feldman and Ballard 1982].

In the HT, features (in the transform space) produce votes for parameters with which the features are compatible. After the voting process, the cell with the most accumulated votes indicates the parameters explaining (consistent with) the most input evidence. HT implementations usually use an N-dimensional array that accumulates votes in discrete cells of N-dimensional parameter space. Then HT interpretation usually consists of finding the winning cell (or cells) by searching the array for local or global maxima using more or less complex algorithms. If the array is considered as a histogram, then the maxima (peaks) correspond to its modes.

In an effort to avoid the space requirements inherent in an N dimensional accumulator array, we at the University of Rochester [Brown and Sher 1982; Brown 1983] proposed to implement vote accumulation and peak-finding (mode estimation) in HT with a content-addressable cache (a hash-table is a software equivalent). The cache is smaller than the full accumulator array, but it must be flushed when its capacity is filled to make room for more votes. The hope is that in a space sparsely filled with votes the cache reliably finds the mode using much less storage then the array. Preliminary experiments with a simulated cache and both simulated and real data were undertaken, and are reported in [Brown and Sher 1982], and summarized in Section 3.4.

I here propose a cache-flushing technique and associated architecture that may significantly improve the performance of cache-based sequential mode-estimation schemes. The basic strategy is inspired by an iterative mode-estimation scheme in the statistical literature [Robertson and Cryer 1974], which uses spatial contiguity (intervals on the real line). The Robertson and Cryer algorithm is similar to the "converging squares" mode-finding algorithm [O'Gorman et al. 1983; O'Gorman and Sanderson 1983]. The new flushing algorithm takes into account contiguity in parameter space; the old flushing algorithms did (indeed, could) not. The new scheme needs more cache complexity than the old, and its logic is more complex. However, It seems the added complexities are not ruinous, and are perhaps amenable to hardware implementation. The new scheme generalizes most mode-finding algorithms in that it can find multiple modes.

Section 2 outlines the problem and the proposed solution. Section 3 is a list of ideas that are more or less related, and an assessment of their utility. Section 4 explores the proposed scheme in more detail. Some technical details appear in Section 5.

Codes
Avail and/or
Dist   Special
A-1

2. The Problem and Proposed Approach

This section is meant to provide context for the basically bottom-up organization of the remainder of this report. One mode-estimation strategy takes samples from a one-dimensional density function, sorts them, and then iteratively constructs ever smaller intervals, each the smallest in the last containing some number of samples determined by the sample size [Robertson and Cryer 1974, and Section 3.3]. This basic converging strategy easily extends to many dimensions, though this may affect the technical results on convergence, consistency, etc. The iterative convergence behavior is achieved also by the converging squares technique of [O'Gorman and Sanderson 1983].

A convergence algorithm for sequential sampling was proposed by Hall [1982]. The work in this report is an implementation inspired by Hall's suggested approach, filtered through some current computer science ideas and existing implementations, such as Cache-based implementations of the Hough transform.

A cache-based Hough scheme [Brown and Sher 1982; Brown, Curtis, and Sher 1983; Section 3.4] maintains a content-addressable cache of vote tallies for parameter vectors. Content-addressability means that geometric relations (e.g. contiguity) between vectors are not necessarily mirrored in the cache data structure. The algorithm of Robertson and Cryer thus differs from current Cache-Hough schemes in two important ways (and some unimportant ones such as requiring absolutely continuous pdfs for convergence proofs).

1) Their algorithm assumes a fixed sample size, and that all the sampling is completed before it runs.

2) It is based on spatial contiguity (intervals), over which it constructs density measurements.

In both these respects the convergence algorithm has an advantage over Cache-Hough, which must deal with samples collected sequentially and which (so far) has no notion of geometric contiguity and hence of vote "density" over a finite area. The first difference is fundamental. It is the purpose of this report to address the second difference and endow Cache-Hough with a flushing strategy guided by geometrical contiguity. The resulting cache-management algorithms resemble existing convergence algorithms to an extent, especially as regards the use of intervals converging on the mode. They are much closer to the suggestion of Hall [1982].

The idea is to use a cache version of quad (oct,...2^d) trees to guide flushing of the highest-resolution tally cache (HRC). In the lower resolution caches (LRCs), each tally entry corresponds to a $2 \times 2 \times ... \times 2$, d-dimensional hypercube of the next higher-resolution cells or parameter vectors, and contains the total number of votes in the corresponding higher-resolution hypercube. In the cache version, only cells and vectors with non-zero counts are explicitly represented. Since the vector components are quantized and thus discrete, I shall usually refer to vector (HRC) entries as cells. The hope is that flushing (and perhaps thereafter barring) votes from contiguous hypervolumes of low voting strength renders mode-capturing more robust. (This hope is so far unsubstantiated by experiment, and arises from intuition only.)

In the proposed scheme, a cache version of a 2^d tree is constructed as votes come in to the HRC. The LRCs implementing lower-resolution levels are updated to be consistent with the HRC. Flushing is triggered by a full HRC, but emanates from some LRC, say FlushCell, whose vote count is low, say FlushCount. Vote counts in LRCs of lower resolution than FlushCell are decremented by FlushCount, and higher-resolution cells in FlushCell's hypervolume are completely removed from the cache. Thus the flush proceeds in both directions, decrementing in the lower-resolution direction and removing in the higher-resolution direction.

Current Cache-Hough schemes only have the HRC, and flush on the basis of low individual tallies (optionally using random mercy to preserve a fraction of low tallies). Single-resolution, content-addressable caches are limited to such "pointwise" flushing strategies. The hierarchical cache captures geometrical structure in the data for use by cache-maintenance algorithms. It provides some concept of vote density, can support a version of accumulator space smoothing, and can implement a flushing algorithm that captures the advantages of convergence schemes without accepting all their limitations. The known convergence algorithms concentrate on finding a single peak. This algorithm penalizes low counts but allows for multiple peaks to survive in the cache, thus implementing multiple mode-finding.

3. Related Work

The ideas in Section 2 suggest various results from statistics and computer science. This section mentions several, and passes judgement on their utility in this context. Briefly we conclude the following.

1) Convergence algorithms for mode estimation seem at this writing to be the most directly relevant statistical methods, though they are most at home in unimodal situations. Other problems, such as PDF estimation and the maximum of a sequence problem, are not as relevant.

2) No technical results about the exact problem of mode estimation with finite memory and a discrete sample space have been found in the literature.

3) Cache Hough methods have promising performance, but suffer from severe myopia when flushing. The incorporation of geometrical contiguity information may help.

4) The usual management of multi-resolution data structures (such as Quad (oct...) trees, Dynamically Quantized (DQ) spaces, and DQ pyramids) is not suited for this application.

3.1 Nonparametric Multivariate PDF Estimation

One natural idea is to estimate not just the mode of the histogram embodied in the accumulator array, but to estimate the corresponding PDF [Wegman 1972, 1982]. PDF estimation is inherently harder than mode estimation, because there is more information to be gleaned from the same input. In fact, some PDF estimation assumes the mode is known. N-dimensional (i.e., multivariate) PDF estimation is

thought in the statistical community to be quite difficult and to require many samples because N-dimensional spaces are (hyper)voluminous. Another difference is that usually a PDF is to be estimated from a fixed-size, completely-gathered sample, rather than from the sequential samples that arise in cache methods. Because of the greater inherent difficulty and the seeming lack of relevant methods, nonparametric PDF estimation does not seem to be an attractive alternative to mode-finding.

Often data is neither purely parametric nor completely non-parametric, and partial information about the form of the underlying PDF can be used to advantage [Sager 1983]. Translating to HT terms, Sager advocates ordering N-dimensional cells by their counts and then applying conventional (one-dimensional) rank statistics. In other words, he would use vote count contours in n dimensions to help describe the PDF. This suggestion appeals to me because sometimes we know the shape of the vote count surface [Brown 1983] and it could be used to help locate peaks. However, again, the resulting algorithm is not for sequential data.

3.2 The Maximum of a Sequence

The sequential nature of cache-based mode finding leads one to associate it with the Maximum of a Sequence (MaxSeq, Secretary, Beauty Contest, etc.) Problem. The pure version of MaxSeq is: you are presented n face-down slips of paper, on each of which is written an integer, and you are to turn them up sequentially until you decide that the current slip has the maximum integer. What strategy should you use to maximize the probabilty of choosing the slip with the maximum integer? It is surprising that even under such seemingly underconstrained conditions an optimal strategy exists and gives you the respectable and elegant winning probability of $1/e$. MaxSeq has been extended in several ways, including allowing a finite buffer of candidates, searching costs, call backs, and so forth [Gilbert and Mosteller 1966; Smith and Deely 1975; Lorenzen 1979, 1981].

Loui [1983] investigated the application of these results to cache-based mode-finding: the conclusion is that MaxSeq is a different problem. In HT mode-finding terms, MaxSeq assumes that all the votes are in and saved, that the final tallies are sequentially presented, and that there is no recalling a previously dismissed tally. The non-recall constraint is especially stringent and non-intuitive in the HT context, and again there is the basic difference that in cacheing the votes come in sequentially, whereas in MaxSeq the totals are in and they (not individual votes) are presented sequentially.

3.3 Mode Estimation

A common method of estimating the mode of a continuous unimodal distribution from n samples is (in 1-D) to sort them and then find the shortest interval containing some number $h(n)$ of them. Then the mode is taken to be some point in that interval (e.g. its midpoint, or the mean or median of the samples it contains) [Venter 1967]. The convergence scheme of Robertson and Cryer is designed to lend robustness to finding the mode of "contaminated" distributions. It refines the interval iteratively, at each stage finding the shortest subinterval containing $c^{(t)}(n)$ samples, for $t = 1,2,...s(n)$ stages [Robertson and Cryer 1974]. Thus the interval is cut

down by a fraction of $[(c^{(t-1)}(n)-c^{(t)}(n)]/[c^{(t-1)}(n)]$ on iteration t. In all these cases, the technical statistical results have to do with choices of $h(n)$, $c(n)$, and $s(n)$ that yield consistent and quick convergence, and with asymptotic distributions. Experimental results of Robertson and Cryer using a $c(n)$ of approximately $2n/3$ to $3n/4$ (here n is the number of samples in the interval being refined, not the total number of samples), indicate that with outliers (noise) or contaminated data (multimodal or in their case a mixture of two distributions) the intervals must start smaller and converge faster to avoid getting confused by local maxima. They recommend that $c(n)/n$ be significantly smaller than $1-p$, p the fraction of contaminated data.

The mathematical restrictions on these methods that must be imposed to allow analytic results are fairly severe, but the strategies are clear and appealing and extend to multiple dimensions. They inspired the approach proposed in this paper.

Finding the shortest interval containing a given number of samples requires search. In statistical models, the samples are from a continuum, and hence will be duplicated only with probability zero. In the accumulator array search, the samples are discrete, and the cells can have more than a unit count. Thus the search will have to keep an updated sum of counts in a (d-dimensional) rectangular volume, which requires slightly more computational effort than merely counting single samples. Fast techniques exist for running rectangular averages, however.

The iterative search of convergence methods is not more expensive than one-time search. To compare Robertson and Cryer's approach to that of Venter, both presume n 1-D sorted real data. The search for the smallest interval containing $h(n) = c^{(t)}(n)$ of them requires comparing the lengths of $n - h(n) + 1$ intervals (thus $n-h(n)$ comparisons). Robertson and Cryer point out that in the iterative scheme, the number of intervals to compare is

$$\sum_{i=1}^{t} [c^{(i-1)}(n) - c^{(i)}(n)] = n\text{-}c^{(t)}(n) = n\text{-}h(n).$$

If the fraction by which the interval is diminished on each iteration is constant, write it as r. Then the above result is derivable from geometric series summation, by which it generalizes to d dimensions. Surprisingly, the iterative search in d dimensions takes fewer comparisons than the one-time search in d dimensions. (Remember, this is the number of hypervolume densities (or total occupancies) that must be computed to find the densest interval. It does not include the number of operations needed to compute each total. For that operation, fast running-total algorithms exist [Rosenfeld and Thurston 1971; Narendra 1978].) In the one-time search, the number of hypervolume densities in d dimensional MxMx...M space that must be computed to find the densest hxh...xh hypervolume, $h = r^t M$ is

$$C_1 = (M\text{-}r^t M)^d = M^d (1\text{-}r^t)^d$$

(Here t is an honest exponent, not an index.) In an iterative search, the number of hypervolumes to be considered is

$$C_2 = (M\text{-}rM)^d + (rM \cdot r^2M)^d + (r^2M \cdot r^3M)^d + \dots$$
$$= (1\text{-}r)^d (M^d + r^dM^d + \dots + r^{(t\text{-}1)d}M^d)$$
$$= M^d(1\text{-}r)^d (1 + r^d + r^{2d} + \dots + r^{(t\text{-}1)d})$$
$$= M^d (1\text{-}r)^d(1\text{-}r^{td}) / (1\text{-}r^d).$$

The ratio C_2 / C_1 is the fraction of comparisons that the iterative search must make compared to the one-time search. The behavior of this ratio is not obvious from the formulae, although for small r it is approximated (from below) by

$$(1\text{-}dr)/(1\text{-}dr^t).$$

Table 1 gives values of the ratio for relevant r, d, and t. It shows that as dimension, the size of the ratio, and the number of iterations go up, the number of density comparisons falls off.

d	t: 2	4	8	t: 2	4	8	t: 2	4	8
2	0.680	0.605	0.600	0.556	0.378	0.336	0.510	0.275	0.175
4	0.411	0.323	0.318	0.210	0.086	0.068	0.140	0.026	0.009
8	0.168	0.103	0.100	0.039	0.007	0.004	0.013	0.000	0.000
		r: .25			r: .5			r: .75	

Table 1: The fraction of hypervolumes that must be compared in an iterative search for the densest, compared to a non-iterative search.

In the converging squares algorithm [O'Gorman et al. 1983; O'Gorman and Sanderson 1983], a space of size nxn (in two dimensions) yields four smaller overlapping spaces. In 2-D, the new spaces are the (k-1) x (k-1) squares in the four corners of the old space. The single (k-1) x (k-1) square of maximum density is chosen for expansion at the next cycle. The common area between the overlapping spaces allows it to be disregarded in computing the differences in density, resulting in substantial computational savings.

Comparing the computations needed for converging squares to two other simple mode-finding algorithms:

-- maximum value: $C(n^2\text{-}1)$

-- smoothing (four points) then maximum value: $3An^2 + C(n^2\text{-}1)$

-- converging squares: $A(n^2+7n\text{-}22) + C(5n\text{-}7)$

where C is a conditional operation, A is an addition operation. Typically a C takes twice as long as an A, and an implementation of converging squares is in fact about three times faster than maximum value and six times faster than smoothing on a VAX 11/780.

3.4 Cache Hough Implementation

The performance of Cache Hough schemes under a variety of conditions (noise, cache length, length of vote bursts, image scanning order, flushing strategy) is tested in [Brown and Sher 1982]. These experimental studies are not backed by formal analysis.

The cache model is that of a single-resolution tally cache (the HRC of Section 2), flushed by either of two strategies. In "Slaughter of Innocents" flushing, all tallies below a threshold are flushed. In "Draft Lottery" or "Random Mercy" flushing, a fraction of all tallies below a threshold is selected at random and flushed. The performance of a cacheing scheme is measured by the ratio

$$SNR3 = \frac{\text{(votes for the vector known to be correct)}}{\text{(maximum votes for any incorrect vector)}}.$$

There are few qualitative surprises in this work. Performance improves with increasing cache length and falls off with increasing noise and fraction of incorrect votes in a vote burst. Scanning strategies seem equally matched except for random with replacement, which may have been prejudiced by relatively small sample size (400 samples (vote bursts) from a 20 x 20 array of features). The lottery flushing strategy works better than the slaughter strategy. Figure 1 shows some sample results.

After it fills (which can be after a few features cause vote bursts), the cache is continuously flushing. Since the content-addressable cache does not maintain contiguity information, a low tally from an active (dense) region of parameter space is as likely to be flushed as a low tally from an inactive (sparse) area. The noise modeled in the experiments is additive noise that does not "spread the peak" in parameter space as does quantization noise. In fact quantization noise is important, and is sometimes taken to be the only important noise effect [Shapiro and Iannino 1979]. It is usually combated by smoothing the accumulator array before searching for modes. Such contiguity-based techniques are difficult and unnatural using only the content-addressable HRC, but become possible in the proposed scheme, which leads to an "urban renewal" flushing strategy in which good neighborhoods are preserved. The analysis of [Brown 1983] shows that when multiple votes are produced for each feature, neighborhoods of high voting strength arise around peaks. Thus the "urban renewal" strategy offered by hierarchical caches seems a promising approach for all known voting schemes.

```
Cache length: 32. Scan: ItoB, LtoR.          Cache Length: 64. Scan: ItoB, ItoR.
Noise: 15%.  Instances: 1.                    Noise: 15%.  Instances: 1.
Parts: 1.  Fpsf length: 9.                    Parts: 1.  Fpsf length: 9.
Part length: 11.  N: 100                      Part Length: 11.  N: 100


Flush: Threshold                              Flush: Threshold

Mean of Ratios:      1.212250                 Mean of Ratios:      1.805774
Std. Dev. of Ratios: 0.972063                 Std. Dev. of Ratios: 0.501593


0.0 •••••••••••••••••••••••••••••            0.0 •
0.2                                           0.2
0.4                                           0.4
0.6 ••                                        0.6
0.8 •••                                       0.8
1.0 ••••••••                                  1.0 ••
1.2 •••••••••••••                             1.2 •••••••••••
1.4 ••••                                      1.4 •••••••••••••
1.6 ••••••••••                                1.6 ••••••••••••••
1.8 ••••••••                                  1.8 ••••••••••••••••••••••••••••
2.0 ••••••••                                  2.0 ••••••••••••
2.2 ••••••••                                  2.2 •••••••••••
2.4 ••                                        2.4 •••••••••
2.6 •••••                                     2.6 ••••••
2.8                                           2.8
3.0 •••••                                     3.0 •
3.2                                           3.2 •
3.4                                           3.4 •
3.6 •                                         3.6
3.8                                           3.8
4.0                                           4.0


Flush: Random Below Threshold                 Flush: Random Below Threshold

Mean of Ratios:      1.578250                 Mean of Ratios:      1.791405
Std. Dev. of Ratios: 0.895189                 Std. Dev. of Ratios: 0.461134


0.0 ••••••••••••••••                          0.0 •
0.2                                           0.2
0.4 ••                                        0.4
0.6 •••                                       0.6
0.8 •                                         0.8 •
1.0 ••••                                      1.0 •••
1.2 •••••••                                   1.2 ••••••••
1.4 •••••••                                   1.4 ••••••••••
1.6 ••••••••••••                              1.6 •••••••••••••••••
1.8 ••••••••••••                              1.8 •••••••••••••••••••••••••••••
2.0 •••••••••••                               2.0 •••••••••••••••••••
2.2 •••••••••••••                             2.2 ••••••••••••••
2.4 •••                                       2.4 •••
2.6 ••••••••••••                              2.6 •••••
2.8 ••                                        2.8
3.0 •                                         3.0 •
3.2 ••                                        3.2 •
3.4 ••                                        3.4
3.6                                           3.6
3.8                                           3.8
4.0                                           4.0
```

Figure 1: Sample SNR3 histograms for four configurations of cache HT, showing the beneficial effects of increased cache length and flushing with random mercy. SNR3 > 1 means the correct parameter vector received the most votes. The bimodality of the distributions is unexplained--the peak at SNR3 = 0 represents trials in which the correct vector was not even in the cache after the HT.

3.5 Quad Trees and DQ Methods

Multi-resolution approches to image understanding and processing have been popular and useful for a long time [Kelly 1971; Warnock 1969]. Keeping the multiple resolutions explicitly in a pyramid data structure has also proven quite useful [Samet 1980; Tanimoto and Pavlidis 1975]. When the resolution pyramid is made of predefined cells, they are usually split symmetrically along each dimension, and the resulting structure is called a quad (oct,...) tree. I call them 2^d trees here. When the cells are split asymmetrically, or the density of resolution varies over a pyramid level, especially when the splitting varies as the contents of the data structure arrive sequentially, a Dynamically Quantized (DQ) space or pyramid results [O'Rourke 1981; Sloan 1981].

A control program usually adapts these data structures to high-resolution data by generating new cells where data is densest. In 2^d trees, this is done by splitting the lower-resolution cells. In DQ spaces, the data structure is a k-d tree [Bentley 1975], and cells are split and merged as data arrives sequentially. Ir 'Q Pyramids, the number of cells at a level is fixed but their extent is varie· y moving the d-dimensional "crosshairs" that split each level into 2^d. Usually th 'nagement of the data structures has the goal of producing cells with equal comp ·y, or numbers of counts. The density of the data is thus mirrored in the data str re (dense data in a region produces a tree that is deeper for that region, for instar iis approach is natural in a sense--it does not require search if enough inforn ,ii is kept in the cells to allow splitting and merging. The data structure and some ancillary information is sufficient to reconstruct the approximate density of the original data, which is inversely proportional to the size of the cells.

DQ structures in the literature suffer from a few difficulties. The cells in DQ spaces can be unintuitive (Figure 2). The splitting and merging algorithms are complex. The cells contain only approximately correct counts after splitting and merging, operations which are controlled by total counts and by count gradient information within a cell. The high-resolution parameters (locations) of counts are lost. DQ Pyramids, since the numbers of cells is fixed, have simpler algorithms, but again produce wrong counts as the crosshairs are moved away from their original positions by adaptive warping as data comes in. Despite all this, the DQ structures seem to be practically usable for some applications, including HT accumulation.

The usual count- (or complexity-) equalizing control strategies for hierarchical data structures have a dual sort of effect from the one we desire, although it is possible to imagine working with (i.e., around) them. In the cache mode-estimation application there is one cache entry per cell, and it would be best if modes were captured inside single cells instead of distributed across several. Also the possibility of flushing everything but one cell (at some level) is attractive. Thus the data structures of dynamically quantized structures are useful, but the management algorithms are inherently difficult and in any case can be modified to match our purposes better.
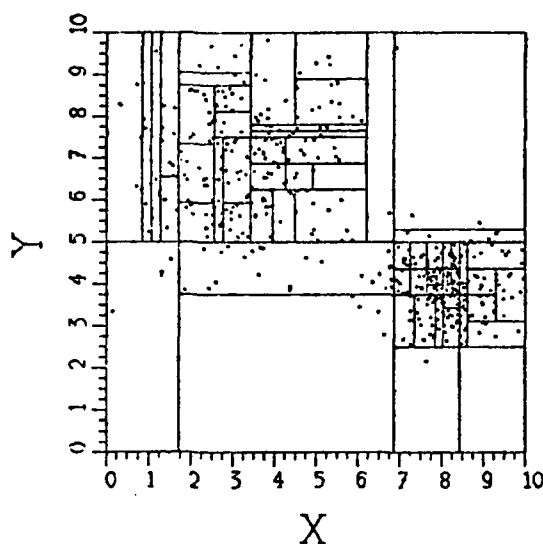
Figure 2: A DQ Space with two modes from [O'Rourke 1981]. The cells in the space result from splitting and merging as data arrive.

4. The Data Structure

4.1 DQ 3^d-Tree

The DQ 3^d Tree is a DQ 2^d Tree (Pyramid) modified to be more useful for our purposes. It is a natural data structure to associate with the convergence mode-finding algorithms. The idea is simple, and shown in Figure 3 in the 3^2 (two-dimensional) case: Construct cells that contain and converge on dense areas, rather than splitting dense cells. The search necessary to create these structures is related to that analyzed in Section 3.3. I do not seriously propose this structure for implementation since adaptive warping would raise all the problems encountered in standard DQ Pyramids, but offer an approximate and presumably simpler version in the next section.
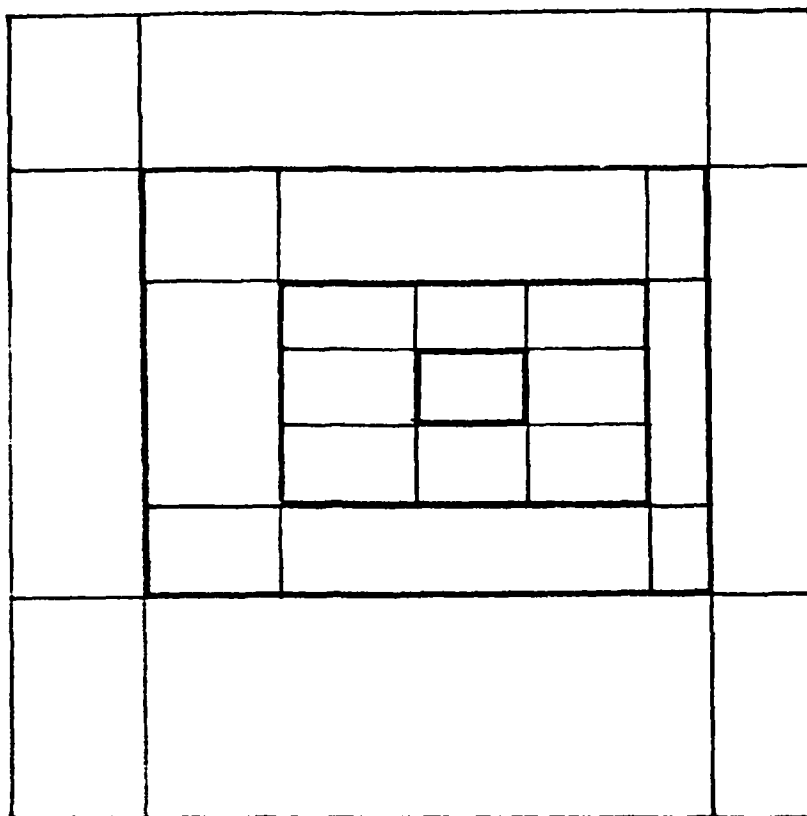
Figure 3: The DQ-3^d Tree. In two dimensions, the DQ non-tree. Central cells contain and converge upon dense areas of data. Non-central cells are candidates for flushing from a cache. Multiple modes simply require splitting a non-central cell at some level.

4.2 Unphased 2^d Tree

As an approximation to the DQ 3^d Tree, consider the Unphased 2^d Tree (Fig. 4). It is simply a 2^d tree augmented at each level by a phase-shifted version of the cells. For definiteness, call the usual cells the U cells, and the shifted ones the S cells.
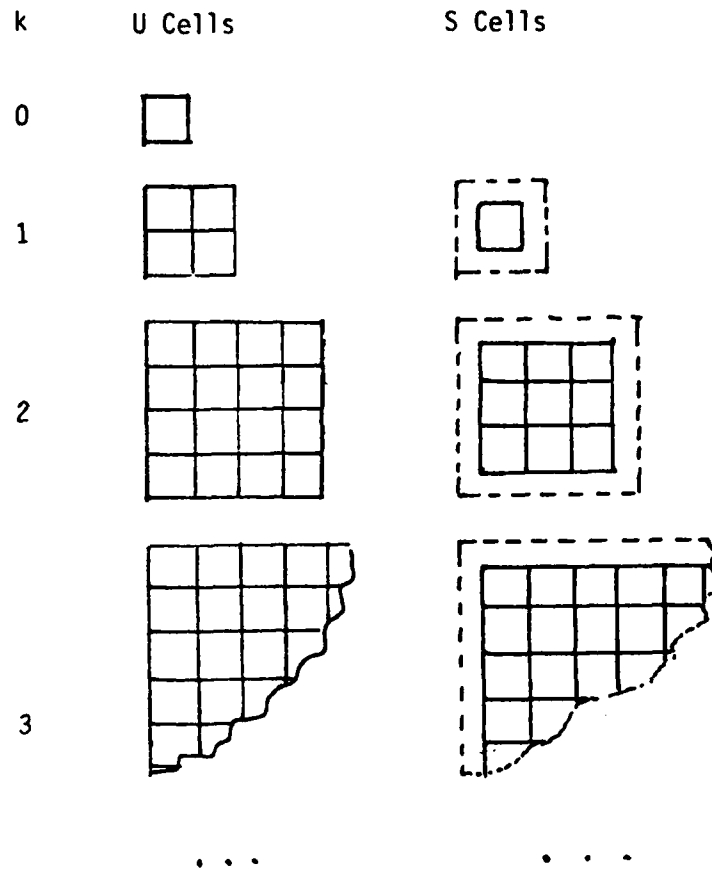
k	U Cells	S Cells

Figure 4: The unphased 2^d tree. In 2-D, the unphased quad tree. The kth layer consists of 2^{kd} U (usual or unshifted) cells and $(2^k-1)^d$ S (shifted) cells. The usual cells are augmented by those shifted half a cell size. The size of shift depends on the level. Their sub-cell inclusion rules are more complicated than for U cells.

This compromise allieviates (but does not cure, unfortunately) the problem of dense areas that lie on predetermined boundaries in the tree, at a considerable computational saving over the (better) solution offered by DQ 3^d Trees. A vote for parameter vector increments the count of a sequence of cells, namely all those cells containing the vector. Each such U cell may be addressed by the highest-order componentwise bits of the last. This produces the normal 2^d tree structure. The construction of the out-of-phase cells is treated in some detail in Section 5.1.

The shifted cells are some insurance against splitting a peak over several cells, as is guaranteed to happen in traditional multi-resolution schemes. We do not want this to happen: it loses a resolution level. Worse, it can make the new cells (each with only about 2^{-d} of the votes for the mode) vulnerable to flushing. Thus, both non-traditional tree management and phase-shifted cells may have a more important effect than just a gain of resolution in the context of cache-based schemes.

5. Technical Details

5.1 Vector Addresses and Arithmetic

The 2^d tree is implemented as a set of separate but communicating caches, one per level for each of S and U cells. The Vector addresses have a number of bits that increases by d with each level of increasing resolution. I propose to use a straightforward translation for the address of U (natural) cells in the 2^d tree, based on their Cartesian coordinates in d-space. The S cell locations in natural, Cartesian coordinates do not have the elegant leading-bits relation with their underlying cells, and so are transformed into T addresses that do. U and T addresses must be differentiated.

The following discussion relates to U cells. In order for a parameter vector (address, d-dimensional location) to be related to its 2^d tree address, it is represented as follows. If x is a d-vector of m-bit quantities (m = $\log_2 M$)

$$x = \begin{array}{llll} x11 & x12 & x13 & ... & x1m \\ x21 & x22 & & ... & x2m \\ xd1 & xd2 & & ... & xdm, \end{array}$$

where the xij are bits. The write x as the single bit string (dm vector)

$$y = x11 \ x21 \ ... \ xd1 \ x12 \ x22 \ ...xd2 \ ...xdm.$$

In other words, read the above array of bits out columnwise. Thus the d high-order bits come first, and last come the d low-order bits. We shall need one bit to distinguish U LRC addresses from T LRC addresses. The final form of address is

$$address = \{U/T\} \ y \ .$$

Let the HRC be assigned level m and the lowest resolution cache entry (a single entry counting the total votes in each cache) have level 0. Then at level k, 0<k<m, x's parameter vector (address) is the bit string of length kd (interpreted as a d-vector of k bit quantities)

$$RightShift(y, \ d(m-k)).$$

Now consider S cells (Fig. 4), which introduce considerable complication. They are shifted by a different amount on every layer. The kth layer of the 2^d tree has 2^k U cells in it. That layer has $(2^k -1)^d$ S cells of the same size. To generate a unique vector address (the T address) for the S cells, I subtract half their linear dimension from their cartesian (U) addresses. (Think of sliding the 3x3 S cells in layer 2 of a quad tree down so they cover the "lower left" 3x3 square in the 4x4 array of U cells.) This is to generate a unique address for the S cell--all its members will now have T addresses with identical leading bits (kd of them at level k), just like the U addresses.

In a natural way, each U cell on any level k has associated cells on all other levels. They are the cells of higher k whose (hyper)volume it contains and the cells

of lower k that contain it. The rule associating U cell addresses at level k_1 with a cell address y at level k is the following.

R1) If $k_1 > k$, all cells at level k_1 having addresses whose high-order kd bits are the same as y are associated with (lie within) the k-level cell at y.

R2) If $k_1 < k$, the single cell at level k_1 whose address is the first $k_1 d$ bits of y is associated with (includes) the k-level cell at y.

S cells are not so simply associated with each other, since the amount of offset varies from layer to layer. However, an S cell on the k layer is made up of 2^d U cells on the k + 1st layer, and it is this correspondence that is used in the flushing strategy.

Figure 5 shows the U, S, and T coordinates of cells (and the association between U and S cells) in a "two-tree," where d = 1.

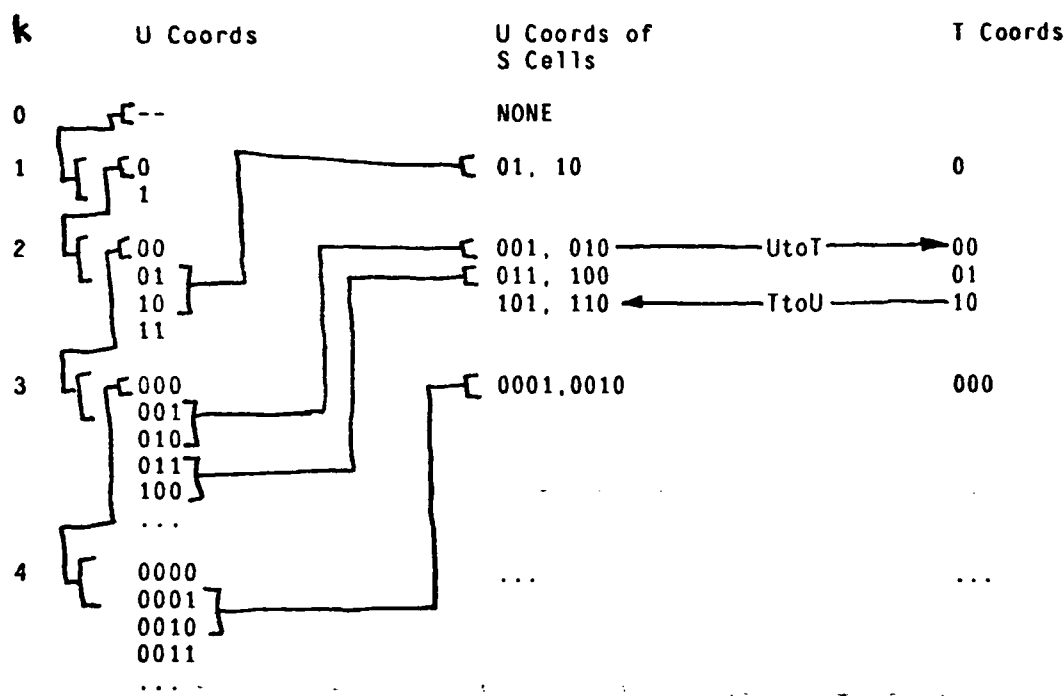| k	U Coords	U Coords of S Cells	T Coords
0	--	NONE	
1	0 1	01, 10	0
2	00 01 10 11	001, 010 ——— UtoT ——► 00 011, 100 01 101, 110 ◄—— TtoU ——— 10	
3	000 001 010 011 100 ...	0001, 0010	000
4	0000 0001 0010 0011 ...	...	...

Figure 5: U, S, and T coordinates. The connected brackets show association relations between U cells and other U and S cells. These 1-D coordinates and transformations extend, componentwise, to d dimensions. UtoT and TtoU transformations are given in the text.

Figure 5 shows that two (in d dimensions, 2^d) cells at level k + 1 are included in a cell at level k. The transformation UtoT maps U addresses at level k + 1 to T addresses at level k. TtoU maps T addresses at level k back to U addresses at level k + 1.

UtoT(k,d): Subtract $2^{(d-k-1)}$ (k leading 0's and a 1) from the $k+1$-bit U address. The leading k bits are the T address.

TtoS(k,d): Take the k bits of T address, append to them the two (in d dimensions, 2^d) possible configurations of one (d) bit(s). Then add $2^{(d-k-1)}$ (k leading 0's and a 1). The leading $k+1$ bits of the resulting two (2^d) addresses are the U addresses.

UtoT and TtoU are easily extended to operate on the linearized dM-vectors in d dimensions. Use the usual truth tables for addition and subtraction (Fig. 6).

```
U  XXXXXXX    XXXXXXX        ...     XXXXXXX   1010110   1011010

Borrows                       ...     0100001   0100101

Subtract Componentwise                                    1111111
-----------------------------------------------------------------
Discard                                                   0100101

T                             ...               1110011
```

```
================================================================
```

```
T  XXXXXXX    XXXXXXX        ...     XXXXXXX   1110011

Append, as one of 2**d choices                           0100101

Carries                       ...     0100001   0100101   .

Add Componentwise             ...                         1111111
-----------------------------------------------------------------
U                             ...               1010110   1011010
```

```
================================================================
```

X	Y	X-Y and X+Y	Borrow	Carry
0	0	0	0	0
0	1	1	1	0
1	0	1	0	0
1	1	0	0	1

Figure 6: Addition and subtraction of address vectors. Simply a carry-ripple operation done bitwise (the ith bit in the each block belongs to the ith dimension's coordinate.) The carries and borrows alone are added and subtracted since the leading bits of the addend and subtrahend are 0. "Overflow" here results from using illegal operands (trying to compute T addresses for U addresses that do not have them, or vice versa).

A more elegant scheme for accessing the shifted cells might result from a more sophisticated coding scheme. Other space-filling addressing schemes are possible and are potentially useful. The Generalized Balanced Ternary scheme is one such, using hexagonal cells in 2-D, truncated octahedra in 3-D, and in general $n+1$ permutahedra in n-space [Gibson and Lucas 1982]. We hope to pursue these topics as time allows.

5.2 Flushing Algorithms

Two strategies for flushing the caches seem useful. The first is called static, and seems primarily useful for uni-modal accumulators and data that comes in from a static situation (for example, a random scan of a single image). The second is called dynamic, and seems more suited for multi-modal data or data from changing sources (time-varying images or raster scans). Both flushes are initiated by conditions in the HRC, usually that it is close to filling up. In both flushing strategies, S or U LRCs with low counts are found and flushed.

To flush a U cell at level k, use UFlush(ACell): remove all ACell's entries. Decrement the count of the cells including ACell in lower-k U cells by ACell's count. Remove entries in higher-resolution (higher-k) cells that are included in ACell--whose leading address bits agree. Every time a U cell at level $k+1$ is flushed, decrement its associated S cell at level k.

To flush an S cell at level k, use SFlush(ACell): remove its entries and UFlush its associated U cells at level $k+1$. This latter flush works its way up and down the hierarchy, flushing and keeping S and U counts consistent.

Flushing could trigger other flushing, as lower-k cells may become flushable through higher-k flushes. In the dynamic algorithm, flushing is all that happens. In static flushing, information is recorded about which cells were flushed, and no new votes in those areas are accepted. This can be implemented several ways, using filter registers to check on the addresses of incoming votes. The registers can contain acceptable or unacceptable ranges of addresses to be checked before votes are inserted in the HRC.

5.3 Number of Lower Resolution Cells

How many lower resolution cells will there be? We can easily put upper and lower bounds on their number, and can appeal to statistics for some more intuitions.

If the HRC is 2^m on a side for d dimensions, there are 2^{md} possible HRC cells. There are $2^{(m-1)d}$ possible LRC U cells on level m-1, $(2^{md}-1)/(2^d-1)$ LRC U cells, and a total of $(2^{(m+1)d}-1)/(2^d-1)$ potential cells in all caches. U cells in $m+1$ levels (down to the single cell at k = 0). With increasing d the number of LRC cells approaches the number in the first LRC, or $2^{(m-1)d}$. For example, in the four-level quad tree with 64 HRC cells, there are 85 total U cells and $64 + 16 = 80$ of these are in the HRC and the first LRC. The S cells approximately double the size of the LRC cache. The entire set of U and S LRCs thus is at worst only about $2^{(m-d)}$ as big as the HRC.

If all the HRC votes are in a small area (a single HRC cell), then only $k+1$ cells are allocated in the U cache, and k in the S cache, for a minimum of some 2k cells.

We may wonder how the cache will look between these two extremes. How likely are we to get empty cells that do not appear in the caches? This question is addressed by occupancy statistics [Johnson and Kotz 1977] and urn models [Cohen 1978]. Say we have C cells and v votes. then there are C^v ways to vote into the cells. Suppose we wish to count the number of ways to vote into C cells so that exactly $C - p$ of them are empty (p of them contain votes). For p chosen cells, the number of distinct ways to vote is

$$p! \; \{v,p\}$$

where $\{v,p\}$ is the Stirling number of the second kind (see below). There are (C,p) ways to choose the p full cells, where (C,p) is the binomial coefficient (see below). Thus the fraction of voting trials in which exactly p cells is filled is

$$F \; = \; \frac{p! \; (C,p) \; \{v,p\}}{C^v}.$$

This quantity may be interpreted as a probability if each cell is equally likely to receive votes. If X is the number of occupied cells, $\Pr[X = p] = F$ is known as the classic occupancy distribution. If cells are not equally likely to receive votes, the expression becomes extremely complex [Johnson and Kotz 1977, eq. 3.5]. The equal-probability situation minimizes the expected number of empty cells, and so is a worst case [ibid].

The binomial coefficient (C,p) is $C! \; / \; (C\text{-}p)! \; p!$

The Stirling number of the second kind $\{v,p\}$ counts the number of ways of partitioning a set of v elements into exactly p subsets, none empty. We have

$$\{v,0\} = 0, \; \{v,1\} = 1, \; \{v,2\} = 2^{(p\text{-}1)} - 1, \; \{v,v\text{-}1\} = (v,2), \; \{v,v\} = 1,$$

and the recurrence

$$\{v,p\} \; = \; p\{v\text{-}1,p\} + \{v\text{-}1,p\text{-}1\},$$

which leads to a Pascal's triangle like construction.

v \ p	1	2	3	4	5	6
1	1					
2	1	1				
3	1	3	1			
4	1	7	6	1		
5	1	15	25	10	1	
6	1	31	90	65	15	1

There is a helpful identity for computing occupancy numbers:

$$p! \{v,p\} = (p,p)p^v - (p,p\text{-}1)(p\text{-}1)^v + (p,p\text{-}2)(p\text{-}2)^v - \ldots \quad .$$

As an example, the distribution of full bins after 8 votes into 8 bins (there are 2^{24} possibilities) is approximately (in percent)

Occupied Bins	1	2	3	4	5	6	7	8
% of trials	.0 ...	.04	2	17	42	32	6	.2

Occupancy distributions have been thoroughly studied in the literature [Johnson and Kotz 1977]. They would be expected to occur in studies of cacheing, and in fact were used to formalize the behavior of working sets [Denning and Schwartz 1972]. For the classical occupancy distribution of interest here, a normal approximation is quite good [Vantilborgh 1974].

Certain limit theorems are known for the classical occupancy distribution. For a fixed number of votes v, as the number of cells p goes to infinity, the expected number of empty cells / .es to infinity, the expected number of cells with one vote goes to v, and the expected number of cells with more than one vote goes to zero. As v and p both go to infinity, with $pe^{(-v/p)} \to w$, then the probability that there are t empty cells approaches a limit

$$\lim_{v \to \infty} \Pr[M_0 = t] = w^t/e^w t!$$

Also, if v and p go to infinity, with $v/p \to w < \infty$, then the limit standardized distribution of M_r (the number of cells with r votes) is unit normal, and

$$E[M_r] = p(v,r)p^{-r} (1\text{-}1/p)^{(v\text{-}r)} \simeq w^r/(e^w r!)$$

$$\mathrm{var}(M_r)/ E[M_r] \simeq 1 - w^r [1 + ((r\text{-}w)^2)/w]/(r! e^w)$$

The last two paragraphs deal with limits of expected values in occupancy distributions, not with the distributions themselves. The results in this section may be useful in calculating expected cache occupancy, or at least in lending some basis for order of magnitude calculations should they be desired. It appears that as a practical matter, the allocation of adequate space for the LRC caches might pay for itself in simplification of the management algorithms.

6. Conclusions and Future Work

Real HT data includes the effects of quantization error as well as inherent sidelobes surrounding peaks. In practice, accumulator arrays are usually smoothed to gather local evidence into a point. With only a few modes in the accumulator array, large volumes of it will be subject only to votes from noise. In traditional cache-HT, spatial contiguity is lost, and the above observations do us no good. A vote flushing and filtering strategy that makes use of spatial contiguity seems likely to improve cache-HT performance, and this report proposes an architecture and algorithms for

that purpose. The strategy is based on statistical mode-estimation algorithms, and the data structure is an augmented version of quad (oct,...2^d) trees. The management of the data structure differs from the usual in that the goal is to keep votes in the fewest cells possible, rather than to spread them out evenly between cells.

For a small investment in space, a hierarchical data structure that keeps track of geometric contiguity can be implemented in a cache environment, with vector addresses encoding the inclusion relations between multi-resolution cells. The flushing algorithms for this structure are simple. Matters become more complicated when an ancillary data structure of shifted cells is added to cope with phasing problems (peaks being split across predetermined cell boundaries). Some aspects of the resulting structure are subject to analytic treatment.

One desirable analytic problem that might be feasible is the treatment of discrete sample space mode-estimation with finite memory, and in particular some properties of an iterative technique such as the one proposed here. How often, say, will it fail to find the mode of an analytically tractable distribution? Continuous approximations (say a continuous version of the whole problem) begin to resemble known convergence algorithms.

Dave Sher implemented a software simulator for HRC-only caches [Brown and Sher 1982]. He is now working on a VLSI implementation of a content-addressable tally cache [Sher 1983]. Neither of these implementations incorporates the hierarchical structure discussed here. We have plans to extend the software simulation to hierarchical flushing algorithms. The relation of the complex flushing algorithms to hardware is under study.

The next step is to simulate this hierarchical cache (initially only with U cells). Methods for vote filtering should be developed and tested. Static and dynamic flushing should be tried with various scanning strategies. If hierarchical caches perform significantly better than single-resolution caches, we must investigate the interaction of hierarchical structure with hardware caches under development.

7. Acknowledgements

8. References

Ballard, D.H., "Generalizing the Hough transform to detect arbitrary shapes," *Pattern Recognition 13*, 2, 111-122, 1981.

Bentley, J.L., "Multidimensional search trees used for associative searching," *CACM 18*, 9, 509-517, September 1975.

Brown, C.M., M.B. Curtiss, and D.B. Sher, "Advanced Hough transform implementations," TR 113, Computer Science Dept., U. Rochester, March 1983; *Proc.*, 8th IJCAI, Karlsruhe, West Germany, August 1983.

Brown, C.M. and D.B. Sher, "Hough transformation into cache accumulators: Considerations and simulations," TR 114, Computer Science Dept., U. Rochester, August 1982; *Proc.*, DARPA Image Understanding Workshop, Palo Alto, CA, September 1982.

Brown, C.M., "Bias and noise in the Hough transform I: Theory," TR 105, Computer Science Dept., U. Rochester, June 1982; to appear, *IEEE Trans. PAMI*, 1983.

Cohen, I.A. *Basic Techniques of Combinatorial Theory.* Wiley and Sons, 1978, p. 118ff.

Denning, P.J. and S.C. Schwartz, "Properties of the working set model," *CACM 15*, 3, 191-198, March 1972.

Duda, R.O. and P.E. Hart, "Use of the Hough transform to detect lines and curves in pictures," *CACM 15*, 1, 11-15, 1972.

Feldman, J.A. and D.H. Ballard, "Connectionist models and their properties," *Cognitive Science 6*, 205-254, 1982.

Gibson, L. and D. Lucas, "Spatial data processing using Generalized Balanced Ternary," *Proc.*, IEEE Conference on Pattern Recognition and Image Processing, 566-571, Las Vegas, June 1982.

Gilbert, J.P. and F. Mosteller, "Recognizing the maximum of a sequence," *J. Amer. Stat. Assoc. 61*, 1966.

Hall, W.J., "Estimating the mode of a multivariate density, based on sequential sampling and with finite storage," Unpublished Note, Statistics Dept., U. Rochester, October 1982.

Johnson, N.L. and S. Kotz. *Urn Models and Their Application.* Wiley and Sons, 1977, p. 107ff, 315ff.

Kelly, M.D., "Edge detection by computer using planning," in Meltzer, B. and D. Michie (Eds). *Machine Intelligence 6.* Edinburgh: Edinburgh University Press, 1971.

Kong, A., "The Beauty Contest with a searching cost," Unpublished Working Paper, Dept. of Statistics, Harvard U., 1982.

Lorenzen, T.J., "Generalizing the secretary problem," *Adv. Appl. Prob. 11*, 384-396, 1979.

Lorenzen, T.J., "Optimal stopping with sampling cost: The secretary problem," *Annals of Prob. 9*, 1981.

Loui, R.P., "How fast Hough?," Internal Working Paper, Computer Science Dept., U. Rochester, May 1983.

Narendra, P.M., "A separable median filter for image noise smoothing," *Proc.*, PRIP-78, 137-141, 1978.

O'Gorman, L., A.C. Sanderson, K. Preston, Jr., and A. Dekker, "Image segmentation and nucleus classification for automated tissue section analysis," *Proc.*, IEEE Conference on Computer Vision and Image Processing, 89-94, Washington, DC, June 1983.

O'Gorman, L. and A.C. Sanderson, "The converging squares algorithm: An efficient multidimensional peak picking method," *Proc.*, IEEE Int'l. Conference on Acoustics, Speech, and Signal Processing, 112-115, Boston, MA, April 1983.

O'Rourke, J., "Dynamically quantized spaces: A technique for focusing the Hough transform," *Proc.*, 7th IJCAI, 737-739, Vancouver, B.C., August 1981.

Robertson, T. and J.D. Cryer, "An iterative procedure for estimating the mode," *J. Amer. Stat. Assoc. 69*, 348, 1012-1016, December 1974.

Rosenfeld, A. and M. Thurston, "Edge and curve detection for visual scene analysis," *IEEE TC-20*, 562-569, 1971.

Sager, T., "Some isopleth methods for mapping multidimensional distributions," presented at ONR Workshop on Statistical Image Processing and Graphics, Luray, VA, May 1983.

Samet, H., "Region representation: Quadtrees from boundary codes," *CACM 23*, 3, 163-170, March 1980.

Shapiro, S.D. and A. Iannino, Geometric constructions for predicting Hough transform performance," *IEEE Trans. PAMI-1*, 3, July 1979.

Sher, D., "The Hough chip," Internal Working Paper, Computer Science Dept., U. Rochester, 1983.

Sloan, K.R., Jr., "Dynamically quantized pyramids," *Proc.*, 7th IJCAI, 734-736, Vancouver, B.C., August 1981.

Smith, M.H. and J.J. Deely, "A secretary problem with finite memory," *J. Amer. Stat. Assoc. 70*, 1975.

Tanimoto, S. and T. Pavlidis, "A hierarchical data structure of picture processing," CGIP 4, 2, 104-119, June 1975.

Vantilborgh, H., "On the working set size and its normal approximation," *BIT 14*, 240-251, 1974.

Venter, J.H., "On estimation of the mode," *Annals of Mathematical Statistics 38*, 1446-55, October 1967.

Warnock, J.G., "A hidden-surface algorithm for computer-generated halftone pictures," TR 4-15, Computer Science Dept., U. Utah, June 1969.

Wegman, E.J., in S. Katz and N.L. Johnson (Eds). *Encyclopedia of Statistical Sciences*, Vol IV. Wiley and Sons, 1982.

Wegmen, E.J., "Nonparametric probability density estimation: I. A summary of available methods," *Technometrics 14*, 3, 533-546, August 1972.